Parallelization of an Electromagnetics Code

Donghoon Chen[*] Victor Eijkhout[†] Paul Sotirelis[‡]

May 6, 1999

Abstract

We describe the parallelization process of an electromagnetics (EM) code used for designing and characterising complex three-dimensional monolithic microwave integrated circuits (MMIC). The code constructs a complex symmetric system from element matrix data, and solves the system by a conjugate gradient method. This paper addresses the obstacles to parallelization, and the solutions found.

1 Introduction

High-frequency radar systems for communication, detection and surveillance incorporate MMIC modules which are characterized by high density and substantial geometrical complexity. In most cases these modules are packaged for electrical and/or environmental protection and the resulting 3D structures exhibit a considerable complexity that impedes design and influences electrical performance due to unwanted parasitics. The design of these complex three-dimensional monolithic circuit (MMIC modules) with hundreds of vias and lines interconnecting a large number of active and passive circuit components to a rather large number of radiating elements, has been a problem of critical importance to DoD. The design and characterization of these circuits requires numerical approaches which can fully characterize the excited electromagnetic fields.

We have concentrated on the development of codes with capability of accelerating numerical computations in electromagnetic problems with large computational domains and/or computationally intensive tasks.

Among the existing full-wave numerical techniques, the finite element method (FEM) has demonstrated superiority in solving general high-frequency electromagnetic problems, especially for complex three dimensional geometries. The FEM solves Maxwell's Equations in the frequency domain using large computational volumes and intensive numerical calculations which have to be performed repeatedly at all frequency points of interest. In addition to these problems,

[*]Radiation Lab.,EECS, University of Michigan, Ann Arbor, MI 48109
[†]Department of Computer Science, University of Tennessee, Knoxville, TN 37996
[‡]ASC/HP Wright-Patterson AFB, OH 45433-7802

the numerical solution of Maxwell's Equations results in huge sparse matrices which further limit applicability of the technique. With regards to substantially increasing the computational efficiency of these techniques, we concentrate to improve FEM through code parallelization strategy.

This work leverages existing synergism between DOD and the University of Michigan scientists and engineers conducting large-scale simulations of advanced software systems. In addition, cross-platform portability and software reusability will be emphasized using the Message-Passing Interface (MPI) standard.

The development of scalable software is based on the FEM techniques to sustain the high accuracy capability while, at the same time, solve complex planar circuits including their packages in very short times, allowing for real time simulations. Such softwares can eventually lead to real time design and optimization.

2 Code structure

The code consists of three parts.

1. Element data is read in,
2. the matrix is formed,
3. the linear system is solved.

The element data is generated by a separate program which models/discretizes the computational domain with tetrahedra elements. The element data represent the discretized computation domain with element cells using the position of element in the domain, nodes/edges of element, and edge/node connectivity of each element.

Our aim in parallelizing the code was to make it scalable: increasing both problem size and number of processors such that the data per processor stays constant, the efficiency should stay constant. In particular, processors should not have data proportional to the total problem size, or operations that take time proportional to the global work.

The second and third parts of the program are conceptually largely parallelizable, though there are several obstacles to automatic parallelization.

- There are few instances of loops that are trivially parallelizable. Most of them are the vector operations in the conjugate gradient method.
- The matrix-vector product in the conjugate gradient method has parallelizable writes, but the reads are only parallel in a shared-memory sense; they need considerable effort in the distributed case.
- The matrix generation loop has conflicting reads and write, as wel as a global counter, that are in the way of easy parallelization.

An OpenMP-like, directive based, approach would thus have limited success on this code. Especially in the third point substantial rewriting is needed.

Between the matrix creation and its system solution there is a further interesting point: the creation can be made parallel over the finite elements of the domain, where as the system solution can be made parallel over the edges in the domain. These two types of parallelism can not be mapped one-to-one; there are more elements than edges because of the boundary, and edges typically belong to more than one element. Thus, if we make independent parallel distributions of the elements and edges, elements belonging to different processors may be adding to the same variable, read, matrix row.

3 Problem distribution strategy

In the current parallel code we use a simple partitioning scheme for both the matrix generation and iterative problem solution. Let the number of elements be E, the number of edges N, and the number of processors P, then we create split points

$$0 = e_0 < e_1 < \cdots < e_P = E, \qquad 0 = n_0 < n_1 < \cdots < n_P = N$$

and we assign to processor p the ranges $E_p = \{e_p, \ldots, e_{p+1} - 1\}$ and $N_p = \{n_p, \ldots, n_{p+1} - 1\}$. Thus, processor p will store the vector element and the matrix rows with indices in N_p.

The work is then distributed as follows:

- Processor p reads the element data for elements in E_p;
- Processor p writes those vector components with indices in N_p.

The above-mentioned mismatch between elements and edges can be described as follows.

- For each processor there is a set N'_p of the edges that adjoin the elements in E_p. Processor p has the data to construct, partially, the matrix elements a_{ij} with $i \in N'_p$.
- Since $N'_p \neq N_p$, processor p has to send those elements in $N'_p - N_p$ to their rightful owners;
- Conversely, every processor has to expect elements of its own part of matrix being sent by some of the other processors.

This last communication stage can be described as a 'sparse all-to-all' stage. We implement this by first having a true all-to-all where it is established which processors have data, and how much, for which other processors. Every processor then posts sends and receives for precisely the necessary communications.

4 Matrix generation from element data

Creating the matrix from the element data is done in a loop over the elements. Its basic structure is as follows.

```
C loop over all elements
    DO I=1,NEL
...
C get element data in TAB
      CALL CRUX( ... )
...
C loop over edges of element
        DO J=1,6
          IF( <certain condition> ) THEN
            NOSEG = TAB(J,3)
            IF( <this edge has not been encountered> ) THEN
              EDST(NOSEG,1) = NPTRX
              EDST(NOSEG,2) = NOSEG
              NPTRX = NPTRX + 1
            ENDIF
...
C fill in matrix elements
        DO J = 1,6
          DO K = 1,6
            IF( <some condition> ) THEN
C calculate i & j location in the matrix
              IM = EDST(1,TAB(J,3))
              IN = EDST(1,TAB(K,3))
```

It is clear that the counter `NPTRX` is a serious impediment to parallelization.

We make this loop parallel (but not scalably parallel) as follows.

- First of all we duplicate the loop, and let the first instance create the `EDST` array, while the second creates the matrix.
- We let the loop over the elements range only over local elements, but write into an array of global size (which can be derived from the element data, using one reduction).

```
    allocated( EDST0(global_max) )
...
    do i=my_first_elt,my_last_elt
```

- We replace the call to `CRUX`, which performs some computation, in the first instance of the elements loop by one that only retrieves numbering data

```
      CALL CRUX0(I,tab,GNN,edna,my_first_elt,local_elt_size)
```

and we write into EDST0 as before.

- After the element loop we perform a reduction to find all places in EDST0 that were written

```
    call MPI_allreduce(edst0,edst,
   >  2*global_max,MPI_INTEGER,MPI_SUM,comm,ierror)
```

- and each processors now constructs the correct numbering:

```
    nptrx = 0
    do i=1,global_max
       if (edst(2,i).ne.0) then
          nptrx = nptrx+1
          edst(1,i) = nptrx ; edst(2,i) = i
       end if
    end do
    global_size = nptrx
```

While this solution is fully parallel in the sense that processors never wait for one another outside the reduction operations, in two places it is not scalable:

- Since the arrays EDST0, EDST are allocated of global size, the total storage for the problem goes up proportional to NP.
- In the final loop, each processor performs work proportional to the global size; hence, the run time for this part of the code is constant in the number of processors.

It would have been possible to make this part of the code scalable, but not without incurring necessary changes in the program that generates the data. Since this piece of code is of minor importance, we decided against this.

Filling in the matrix elements has to be made local too. After calculating the location in the matrix, we write the element either in the local matrix, or in a temporary one:

```
     DO J = 1,6
       DO K = 1,6
         IF ( <some condition> ) THEN
           IM = EDST(1,TAB(J,3))
           IN = EDST(1,TAB(K,3))
C Write the matrix element into the appropriate matrix
           if (im.lt.my_first_row.or.im.gt.my_last_row) then
C write into ASQ, INQ1, INQ2
...
           else
C write into ASQ_t, INQ1_t, INQ2_t
...
```

Although we have made a very simple split of the matrix rows over the processors, it is reasonable to expect that elements assigned to one processor will mostly generate the rows assigned to that processor. For the remaining rows, stored in `ASQ_t` et cetera, we need an all-to-all communication stage.

- By inspection of `INQ1_t` we decide which off-processor rows have been partially generated, and to what processors they belong.
- With an `MPI_Allgather` call we derive which processors are going to receive rows from which.
- The rows are exchanged, and the incoming data is merged in with the already existing local matrix.
- The resulting local matrix is then analysed to determine what variables on and off processor will be needed for a matrix-vector product; see below. This communication pattern will be saved, since it also determines how data is going to be sent every time a matrix-vector product is performed.

In the sparse all-to-all of the matrix row exchange there is a considerable amount of data being sent through the network. Apart from the network load, this means that, if we would use non-blocking sends and receives, processors have to allocate a considerable amount of buffer space. To reduce this, we use blocking sends and receives. Since we are dealing with a sparse all-to-all, some amount of sophistication is needed. We orchestrate the communication as follows:

```
do iproc=0,me-1
    < receive from processor iproc >
end do
do iproc=me+1,nprocs-1
    < send to processor iproc >
end do
do iproc=nprocs-1,me+1,-1
    < receive from processor iproc >
end do
do iproc=0,me-1
    < send to processor iproc >
end do
```

The basic idea here is to perform first all sends to higher numbered processors, then all sends to lower numbered processors. This scheme is dead-lock free: processor 0 has no receives in the first loop and will start immediately by sending in the second loop, and correspondingly the other processors start by receiving from it; the last processor has no receives in the third loop, so it starts sending in the fourth loop immediately, and other processors, because of the downward numbering of the third loop, start by receiving from it.

5 Parallel conjugate gradients

Conjugate gradient iterative methods have following components:

- Vector updates; these are trivially parallel.
- Inner products. These involve a global accumulation and distribution step; thus, they are incur a certain communication time.
- Matrix vector product. Under the normal 'owner computes' rule, these have parallelizable writes, but the reads are a problem.
- Preconditioner application; in the current code this is a variant of a Jacobi preconditioner, so it is trivially parallel.

5.1 Parallel matrix vector product

For the analysis of a parallel matrix vector product $y = Ax$ we consider the line

$$y_i = \sum_j a_{ij} x_j.$$

Because of our partitioning scheme, for $j \in N_p$, both y_i and all a_{ij} are present on processor p. The problem is then with those j-values for which $a_{ij} \neq 0$, and $j \notin N_p$; these values have to be sent from the appropriate processors.

Practically, we have to change the matrix storage slightly. The original matrix was stored in a variant of compressed row storage as

```
complex*16 ASQ(imax,n)
integer INQ1(imax,n),INQ2(n)
```

where the arrays are arranged such that

$$a_{\mathtt{INQ1}(i,j),j} = \mathtt{ASQ}(i,j) \qquad \text{if } i < \mathtt{INQ2}(j)$$

We amend the storage scheme by adding an array `INQ2b(n)` such that for all $j \in N_p$:

$$1 \leq i \leq \mathtt{INQ2}(j) \Rightarrow i \in N_p; \qquad \mathtt{INQ2}(j) + 1 \leq i \leq \mathtt{INQ2b}(j) \Rightarrow i \notin N_p$$

This entails a permutation of the elements in each matrix row.

Additionally, we renumber the permuted elements

$$\text{for } 1 \leq i \leq \mathtt{INQ2}(j): \quad \mathtt{INQ1}(i,j) \leftarrow \mathtt{INQ1}(i,j) - n_p + 1$$

so that the indices refer to the numbering of the local array instead of the global numbering.

A similar numbering is performed on `INQ2b`, so that the matrix elements address a second array of right hand side elements. Specifically, for the matrix elements that refer to off-processor x_j-values, we allocate an array of length

$$o_p = \#\{a_{ij} \colon j \in N_p \text{ and } i \notin N_p\},$$

and we renumber the matrix elements with $\texttt{INQ2}(j)+1 \leq i \leq \texttt{INQ2b}(j)$ to index this array.

In order to overlap communication and computation, we arrange the matrix-vector product as follows:

1. The processor posts (MPI `ISend` and `IRecv` calls) communication requests for off-processor values of x_j.
2. The local part of the matrix-vector product is performed.
3. The processor waits for all communication to be finalised.
4. The remaining part of the product is evaluated.

Naturally, each processor keeps lists of precisely the other processors it communicates with.

6 Timing results

In tables 1 and 2 we report execution times in seconds for the three parts of the program:

1. input from file of the element data,
2. construction of the distributed matrix, and
3. solution of the linear system.

We tested two data sets. The small data set has 5460 elements and the matrix is of size 5712; the large data set has 113,520 elements and the matrix is of size 129,163.

np=	1	2	4	6	8
in	2.25	2.1	1.82	1.5	1.5
mat	13.7	4.2	1.7	1.1	.8
solve	153	79.4	39.5	27.5	22.3

Table 1: Execution time for distributed input, matrix construction, and system solution on networked Sparc stations for a small data set.

np=	1	2	4	6	8
in	51.7	48.3	46.2	37.0	36.6
mat	2494	700	325	203	121
solve	22.5k	12.5k	6374	4148	3028

Table 2: Execution time for distributed input, matrix construction, and system solution on networked Sparc stations for a large data set.

We see an almost perfectly linear speed up of the solution process and the matrix construction. The input phase shows some speed-up, but not much, since each processor is having to read the full input files.

The matrix construction shows super-linear speedup, which levels off around 4 processors on the small problem, but continues on the large problem. This is caused by the irregular data access in this phase; for larger numbers of processors the problem will increasingly fit in cache, giving better performance.

7 Acknowledgements

This research was partially funded by the DoD High Performance Computing Modernization Program, ASC Major Shared Resource Center, through Programming Environment and Training (PET). Views, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of Defense position, policy or decision unless so designated by other official documentation.

List of Tables